Spring 3-31-2003

An Efficient Implementation of Query/Advertise ; CU-CS-948-03

Dennis M. Heimbigner
University of Colorado Boulder

An Efficient Implementation of Query/Advertise

Dennis Heimbigner

CU-CS-948-03

31 March 2003

University of Colorado at Boulder

Technical Report CU-CS-948-03
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309–0430

An Efficient Implementation of Query/Advertise

Dennis Heimbigner

31 March 2003

Abstract

It is demonstrated how a publish/subscribe system can be extended to support the efficient distribution of queries to relevant sites. Queries are encoded as messages that are efficiently distributed to sites providing advertisements, which are special queries that describe the data sets available at each site. An important aspect of this research is to provide a sufficiently powerful language for expressing queries. It is shown how adding a form of constraint to the system as a first class class object can support expressive queries. A query system is constructed on top of the Siena wide-area publish/subscribe system, and it is shown how to optimize the distribution of queries.

1 Introduction

A publish/subscribe system is normally used to distribute event notifications to a network of interested *subscribers* based on the content of those notifications. It turns out that it is also possible to use publish/subscribe in an alternate mode in which *queries* are distributed to a network of *advertisers* of data sources. This second use for publish/subscribe, referred to here as *query/advertise*, provides functionality similar to that of many peer-to-peer networks such as Gnutella [6] and Freenet [5].

In a previous effort [7], we demonstrated that a publish/subscribe system could be used to mimic Gnutella, but with improved security, anonymity, and especially efficiency. This experience convinced us that publish/subscribe systems could provide a good substrate on which to implement query distribution. The one flaw in this hypothesis involved query expressiveness. Our initial effort only supported conjunctions of equality queries (e.g., $x = 5 \ \wedge \ y = 3$), which were useful, but we felt that further improvement was possible.

The goal of this paper is to demonstrate how a publish/subscribe system, specifically Siena [3], can be extended to better support query/advertise by providing a useful query "language" for expressing queries. The approach we take is to embed a specific class of constraint predicates as first-class objects into the type system provided by the underlying publish/subscribe infrastructure. These predicates allow us to move beyond equality expressions to support queries involving conjunctions over many kinds of relational expressions.

The paper first describes the relationship between query/advertise and publish/subscribe, and the notion of matching for queries and advertisements. Our query system extends the structures of Siena, so we will describe the notifications and subscriptions provided by Siena. We then discuss the format of constraints, and we discuss the mechanics of modifying Siena to include constraints while also maintaining the optimizations used by Siena for efficient message distribution.

2 Query/Advertise

In the query model, an advertiser is a client of the query/advertise system who "advertises" the availability of some kind of information using a special kind of query that describes the data available at that client. Other clients issue queries that are distributed to each advertiser whose advertisement is deemed to "match" the query. It is the job of the query/advertise system to ensure that queries are efficiently

directed only to those data sources that *may* have information matching the query. This is in contrast to a many peer-to-peer systems in which every query is sent to every data source. It is this behavior that makes such systems so inefficient.

Upon receiving the query, the advertiser applies it to its local data and responds with the resulting data. As described elsewhere [7], the response may be returned through the publish/subscribe network but it may be returned using some other mechanism such as a point-to-point TCP connection. The net effect is that the original query client receives, from multiple sources, data that matches its query. That client can then collate the responses to produce an aggregated result. This whole process involves a sequence of advertise-query-respond combinations, but we will refer to this simply as *query/advertise*.

For comparison purposes, recall that in publish/subscribe systems, clients *publish notification* (or *event*) messages with highly structured content. Other, subscribing, clients make available a filter (a kind of pattern) specifying the *subscription*: the content of notifications to be received at that client. It is the job of the publish/subscribe system to ensure that notifications are efficiently delivered to the clients with matching subscriptions.

Publish/subscribe and query/advertise are in a sense duals of each other. A subscription represents a way for a site to indicate that specified notifications should be routed to the subscriber. An advertisement represents a way for a site to indicate that specified queries should be routed to the advertiser. Similarly, a publisher sends out notifications that should be routed to matching subscribers. A *queryer* sends out queries that should be routed to matching advertisers. As we shall see, this duality is important because query/advertise is mapped onto publish/subscribe by mapping advertisements to subscriptions and queries to notifications.

It is also the case that both query/advertise and publish/subscribe assume an architecture where many clients are connected together via an overlay network of interconnected servers providing *content-based routing* [4]. These routers are responsible for sending copies of messages (events or queries) to all clients exporting matching subscriptions or advertisements.

Query/advertise has the notion of a response that is inherent in any system for querying data sources, but which has no dual in publish/subscribe. Responses may in fact be provided without using the publish/subscribe system at all. Therefore mapping the response mechanism to publish/subscribe requires some special handling. As described in the discussion of Site-Select (Section 8), it is possible (and even useful) to extend publish/subscribe with some special mechanism to support responses.

3 Siena

We explicitly build upon the Siena publish/subscribe middleware system developed at the University of Colorado because it provides a convenient interface and offers important optimizations for improving the efficiency of notification distribution. We will exploit these optimizations to achieve similar efficiencies for queries.

Siena notifications are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of types. In standard Siena, the set of supported types is bool (true or false), long (64-bit integer), double (128-bit floating point), and byte-string, which also subsumes the more traditional string type. An example message could be represented as shown in the left column of Figure 1.

A client establishes a subscription by constructing a filter (a pattern) that specifies the kinds of messages it wishes to receive. A filter is a set of triples of the form (attribute, operator, value). A filter matches a notification if the value associated with each attribute in the notification satisfies all corresponding filter triples that have the same attribute name. That is, for a given filter F and a given notification N, the following holds.

$$\begin{aligned} &\forall\ triples\ (x, op, a) \in F \\ &\quad (\forall\ pairs\ (y, b) \in N \\ &\qquad (x\ =\ y\ \Rightarrow\ Apply(op, a, b) = true)) \end{aligned} \tag{1}$$

Notification	Filter
{(author,"John Steinbeck") (title,"Grapes of Wrath") (edition,1) (instock,true) }	{(author,*,"Stein") (edition,>,1) }

Figure 1: Example Notification and Filter

$$where\ Apply(op, a, b)\ =\ (a\ op\ b)$$

The set of filter triples may be considered to be logically "and"ed together. A logical "or" can be achieved by specifying multiple separate filters. The right side of Figure 1 shows an example filter that would match the message on the left side. Table 1 shows the complete set of pre-defined operators available in standard Siena. Since they are used for matching, they all produce a boolean result.

It is important to note that the attribute names used in messages and filters have no inherent semantic meaning. As with all such attribute-based systems, there must be some external agreement about their meaning, and all parties must adhere to that agreement.

Siena adopts a peer architecture where arbitrary Siena servers connect to form a specific topology. In the simplest case, a client connects to a server and establishes a subscription. The server then forwards the subscription filter to all of its peers. Each peer records where the subscription came from, and forwards it to its peers. Later, when some other client connects to a server and generates an event message, the local copy of the filter can be applied at that server to determine the next server to whom the message should be forwarded. If a message is generated for which no filter matches at the local server, then it will not be forwarded at all and so will generate no inter-server traffic. This kind of content-based routing is analogous to IP routing in the Internet, but instead of specific IP addresses, the content of messages of determines the destination (or destinations) for the message.

4 Query Matching

Independent of the particular chosen query language, the query/advertise system requires two interpretations of a query: *query application* and *query intersection*.

The first interpretation (application) is the conventional one where a query is applied to a data set to produce a result set of data items matching that query.

Table 1: Siena Filter Operators

Operator	Argument Type
Equals (=)	bool, long, double, byte-string
Not-Equals (!=)	bool, long, double, byte-string
Less-Than (<)	long
Greater-Than (>)	long
Less-Equals (≤)	long
Greater-Equals (≥)	long
Prefix (> ∗)	byte-string
Suffix (∗ <)	byte-string
Contains (*)	byte-string
Any (any)	N.A.

Query	Advertisement
{(author,(=,"John Steinbeck")) (edition,(<,2))	{(author,*,"Stein") (edition,=,1) } (copies,>,1) }

Figure 2: Example Query and Advertisement

The second interpretation, query intersection, is used to determine if a query "matches" (is relevant to) an advertisement. Thus, given two queries Q1 and Q2, we say that Q1 intersects Q2 if the following holds.

$$\exists\ datasource\ d\ s.t.\ ((Q1(d)\ \cap\ Q2(d))\ !=\ \phi) \tag{2}$$

That is, there exists a dataset, d, such that the result of applying Q1 to d and the result of applying Q2 to d have at least one data item in common. If Q1, say, represents the advertisement, then it makes sense to send Q2 (the query) to the advertising site because it may be able to provide a result.

In practice there are several things to note.

1. We determine query intersection based on the actual queries, not on any specific dataset, thus there is no guarantee that the specific data set held at some site will actually satisfy equation 2.

2. For more efficient matching, an advertiser may provide several advertisements such that the union of these advertisements represents his whole data set.

3. In order to avoid providing too many advertisements, a site may "fib" and provide an advertisement that technically covers more data than is available at the site. This allows for more "approximate" advertisements.

5 Query Definition in Siena

Our goal is to introduce some form of query expression as a first class data value in Siena. We chose to introduce the triples used in filters as the basis for our query expressions, and we did so because they are expressive, they are easy to use for a user of standard Siena and because they easily integrate into Siena while maintaining many of the desirable efficiencies provided by the Siena infrastructure.

Our specific approach was to introduce a *constraint* data type as a legal value for attribute-value pairs in a Siena notification message. A constraint has the form $(operator, value)$, which is of course a subscription filter triple without the attribute name.

Figure 2 shows a query and an advertisement. Note that the query technically keeps the two-element pair format of a message notification The difference is that the value of the attribute is now a constraint as shown on the left side of Figure 2. We will use the term *named constraint* to refer to such a pair whose value is a constraint.

In this model, query application occurs when an advertising site receives a matching query message. It takes the message and applies some subset of its named constraints to its data source to produce a response. The exact set of named constraints and the exact method of application are defined by the receiving site.

The other interpretation of a query is for query intersection (Section 4). This determines if a given query message is applicable at given site as determined by the advertisements exported by the site. Recall our definition of a match between a filter and a notification as defined in equation 1. There we assume that each triple is matched against each pair with the same attribute name and a match is declared if all these individual matches succeed.

We adapt this match procedure to define query intersection. That is, for a given advertisement A and a given query Q, the following holds if the query matches (intersects) the advertisement.

$$\begin{array}{l} \forall\ triples\ (x, op1, a) \in A \\ \quad (\forall\ named\ constraints\ (y, (op2, b)) \in Q \\ \quad\quad (x\ =\ y\ \Rightarrow\ Intersect(op1, a, op2, b) = true)) \end{array} \tag{3}$$

Note that we substituted the *Intersect* procedure for the *Apply* procedure in equation 1.

So we say that a query and an advertisement intersect if each set of corresponding triples and named constraints intersect as defined by the *Intersect* procedure. This now reduces our task to defining $Intersect(op1, a, op2, b)$ for every pair of operators ($op1$ and $op2$) with arbitrary attribute values (a and b).

6 Defining Operators in Siena

We must digress slightly to discuss the details of operator definition in Siena. The process of adding an operator to Siena involves defining two procedures: *Apply* and *Covers*.

6.1 The Apply Procedure

Equation 1 requires the computation of expressions of the form $Apply(op, a, b)$. Thus adding an operator to Siena requires defining an *Apply* procedure to compute this value.

The Apply procedure for ordinary operators defines the ordinary application semantics of the operator. Thus, given two values a and b and an operator op, this procedure computes the value of (a op b) (e.g., $(5\ >\ 7)$).

6.2 The Covers Procedure

When defining a new operator, the other required procedure is *Covers*. This is required to support one of the forms of scalability provided by Siena. This procedure supports an optimization that can reduce the number of filters that a given server must maintain. Without this optimization, Siena would be forced to propagate all filters to all Siena routers.

The *Covers* relation between two filters F1 and F2 is the key to this optimization. The relation ($F1\ Covers\ F2$) holds if every message that matches F2 also matches F1. In other words, the set of messages matching F1 is a superset of the set of messages matching F2.

Since a filter is composed of triples of the form (x, op, a), F1 covers F2 if the following holds.

1. Each attribute name occurring in F2 also occurs in F1.

$$\begin{array}{l} \forall\ triples\ (x2, op2, b) \in F2 \\ \quad (\exists\ triple\ (x1, op1, a) \in F1\ s.t.\ x2 = x1) \end{array}$$

2. The set of values satisfying a triple from F2 is a subset of the set of values satisfying any similarly named triple from F1.

$$\begin{array}{l} \forall\ triples\ (x, op1, a) \in F1 \\ \quad (\forall\ triples\ (x, op2, b) \in F2 \\ \quad\quad (\forall\ z\ ((z, op2, b) = true\ \Rightarrow\ (z, op1, a) = true))) \end{array} \tag{4}$$

We define a *Covers* procedure with the following interpretation.

$$\begin{aligned} Covers(op1, a, op2, b) &= true \\ &\quad if\ (\forall\ z\ ((z, op2, b) = true \\ &\quad\quad \Rightarrow (z, op1, a) = true))) \qquad (5) \\ Covers(op1, a, op2, b) &= false\ otherwise \end{aligned}$$

(6)

At a given router, the Covers relation forms a forest of partial order graphs over all the filters known at that router. Two filters F1 and F2 are in the same partial order graph if ($F1\ Covers\ F2$) or ($F2\ Covers\ F1$). Otherwise, they are in different graphs in the forest. Siena routers need only propagate the most general filters, which are those that are at the root of each Covers graph.

Again, in order to participate in this optimization, each operator (op) must define the procedure $Covers(op1, a, op2, b)$ to compute if the Covers relation holds between two triples $(x, op1, a)$ and $(x, op2, b)$ from two different filters. This procedure assumes that (1) each triple has the same attribute name, and (2) that the operator in one or both of the triples is operator op.

It is is important to note that the Covers procedure is optional, albeit highly desirable. Defining the Covers procedure to always return false is acceptable. The consequence, though, is that all filters containing that operator will be propagated to all Siena routers and significant inefficiencies may result.

7 Implementing Queries in Siena

The first step in implementing queries in Siena is to introduce a new data type representing constraints. This is straightforward to implement and requires defining a new type in the type enumeration and defining serialization and de-serialization procedures for constraints.

The second step is figure out the effect of our new data type on the *Apply* and *Covers* procedures. Two question arise in this context.

1. When we are matching a message against a filter, how do we know when to compute the normal *Apply* semantics and when to compute the *Intersect* semantics?

2. How do we compute the Covers relationship between two advertisements?

The answer to question 1 is that we need some kind of signal to indicate what to do, but we need to do it in such a way as to minimize the disruption to the standard operation of Siena: we would like to be able to distributes queries and normal notifications using the same set of Siena routers.

We use the presence of a constraint value in the message as our signal to invoke intersect semantics. So assuming that the second argument comes from the notification message, we can define a revised *Apply* procedure as follows

$$\begin{aligned} Apply(op1, (op2, a), b) &= Intersect(op1, op2, a, b) \\ Apply(op, a, b) &= (a\ op\ b) \end{aligned}$$

The second equation is the standard interpretation used for all operators when a constraint is not involved. The first equation is used to invoke intersection semantics.

The remaining task with respect to *Apply* is to define *Intersect(op1,op2,a,b)*. Recall that the idea is to try to find out if there is some value for x that can satisfy both ($x\ op1\ a$) and ($x\ op2\ b$). Table 2 shows *some* examples for pairs of operators; the values of a and b are assumed arbitrary. The first row, for example, says that $(x, =, a)$ intersects $(x, op2, b)$ is true if ($a\ op2\ b$) is true; this is because the only possible value that can satisfy both is a. The second row says that $(x, <, a)$ intersects $(x, <, b)$ is always true because any x < min(a,b) will satisfy both constraints.

Table 2: Intersect() Semantics (Partial)

Op1	Op2	op1 ∩ op2	Rationale
$=$	$op2$	$a\ op2\ b$	(only $x = a$ works)
$<$	$<$	true	(any $x\ <\ min(a,b)\ works$)
$>$	$>$	true	(any $x\ >\ max(a,b)\ works$)
$<$	$>$	$a\ >\ b$	(any $x\ \in\ range(a,b)\ works$)
			...

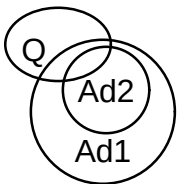

Figure 3: Advertisement Covering

Our last concern is to compute the Covers relation. In the query/advertise context, the Covers relation is being computed over advertisements and the question arises: is the Covers relation for subscriptions directly applicable to Covers for advertisements? The short answer is yes.

To see this, we need to go back to the definition of an advertisement, which is that it describes a data set at a source. Thus, we can say that for advertisements Ad1 and Ad2, Ad1 Covers Ad2 if the data set described by Ad1 is a superset of the data set described by Ad2. Figure 3 illustrates this. If we propagate only Ad1 to other routers, then any query Q that intersects Ad2 will also intersect Ad1, so that the query will get directed correctly.

Since the Covers relationship purely computes the superset relationship, our existing Covers procedure can be used on advertisements to produce the correct result. The only difference is that for subscriptions, the superset relationship refers to the space of notifications and for advertisements it refers to the space of data sets.

8 Related Work

This work is closely allied to the Site-Select system [9] being developed at the University of Virginia as part of the joint Willow project between Colorado, Virginia and UC, Davis [13]. Site select provides a simpler query language based on bit-sets. In effect a client advertises a set of bits that represent boolean properties that characterize the client site. A query is another bit-set whose bits indicate attributes of interest. The queries are directed at the sites that advertise at least the same bits as in the query. By adding, as we have done [8], some bit-set specific operators, we can subsume Site-Select matching. On the other hand, Site-Select has a built-in response mechanism that supports a limited form of aggregation of responses to be returned automatically to the originator of the query. As we have indicated, our query/advertise system is agnostic with respect to how responses are returned. We anticipate that we can merge this effort with the Site-Select response mechanism to produce a more powerful query/advertise/response system.

Resource discovery systems are closely related to query/advertise and can easily be realized using the query/advertise system. This is because many of these systems in effect advertise resources based on descriptive properties that may be queried. Intentional Naming [1] represents some of the earliest work in resource discovery. Its query language was relatively sophisticated and could handle, for example, some

forms of inequalities. Its protocol was strictly oriented to discovery and did not support the equivalent of Covers. Jini^{tm} [10, 21] is perhaps the best known of the resource discovery systems. Jini defines a collection of programming interfaces. The implementations behind them are prototypes that do not appear to have addressed issues such as wide-area scale and message traffic.

Many peer-to-peer systems [14] have the capability to carry out the equivalent of distributed query. This is because most of them are being used for file and music sharing, and it is important to be able to locate music files based on various attributes such as artist, title, and sampling rate. Examples of this include Kazaa [12] and the now defunct Napster [16, 22]. For most of these systems, the properties upon which queries can be built is essentially fixed by the network provider.

Gnutella [6] and Freenet [5] are examples of peer-to-peer systems that are in some ways more general than music sharing networks. The primary problem with Gnutella has been its query distribution protocol. In its original incarnation, it was extremely wasteful of bandwidth because it propagated messages indiscriminately. Attempts have been made to improve Gnutella's protocol [15] , but with limited success. We have demonstrated in our previous work [7] that query/advertise built on publish/subscribe could produce a system that was similar to Gnutella but was superior in performance. With the work described here, the expressiveness of query/advertise is now at least as powerful as Gnutella.

Freenet provides anonymous distributed file sharing. It is based on distributed hash tables, and as a result it has a much more restricted notion of query than any other peer-to-peer system: clients ask for specific files (identified by a unique hash) and the search process stops when that specific file is found. Caching is also supported. Freenet uses message traffic about as efficiently as does Siena's content-based routing, and far more efficiently than Gnutella. It is apparently still an open question [2] if Freenet can be extended to support the general queries provided by query/advertise.

Astrolabe [20] provides yet another model for distributed query. It organizes its network of sites into a tree. Queries are SQL statements that are propagated from the top of the tree down to the leaves. These queries are currently restricted to aggregation queries (e.g., summation, average, and count). The queries are executed at the leaves and the aggregated values are passed up to the next level where they are further aggregated. This is repeated until a single value is computed at the root. As the authors note, new query propagation is assumed to be relatively infrequent; rather the model efficiently ensures that the values of existing queries are maintained in the face of changes in the underlying databases upon which the queries are applied. In contrast, our query/advertise supports dynamic propagation of queries as the norm, but has no ability to support hierarchical aggregation because no hierarchy exists.

Our query/advertise system is built upon Siena, but other publish/subscribe systems are available as alternatives upon which to build a query/advertise system. There are two issues here: scalability to wide-area networks (using some equivalent of the Covers relationship) and expressiveness. Most publish/subscribe systems are designed for local-area network use. Examples are Field [17] and ToolTalk [11]. Some other systems address are intended to operate over wide-area networks. Examples include TIBCO [19], Elvin [18], and Siena. Both TIBCO and Elvin appear to suffer from a lack of automatic Covers relation support. The equivalent of the Covers relations must be manually established and maintained.

TIBCO is also representative of another form of publish/subscribe system often referred to by the term "subject-based." Expressiveness is a problem for such systems because they provide only a single content string (the subject) for use in routing. This severely limits expressiveness, and it is not clear if any sort of reasonable query/advertise system could be built using a subject-based system.

9 Conclusions

We have demonstrated how to modify the Siena publish/subscribe system to efficiently support query/advertise and to support an expressive query language. The distribution is controlled by advertisements describing the data sets available at each site. The query language is supported by adding constraints as a first class data type to the type system of the publish/subscribe infrastructure.

A modified version of Siena is available from the author. This version implements the query/advertise system described in this paper. Further improvements to the query language are being explored. These

include adding variables to allow inter-triple value matching and support for additional queryable values such as unification of functional terms.

10 Acknowledgements

This material is based in part upon work sponsored by the Air Force Research Laboratories, SPAWAR, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-00-2-0608 and N66001-00-8945. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proc. of the 17th ACM Symposium on Operating System Principals*, Kiawah Island, SC, 1999.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *CACM*, 46(2):43–48, 2002.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing*, Portland OR., July 2000.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, Jan. 2000.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000. International Computer Science Institute.

[6] *Gnutella Home Web Page*. http://gnutella.wego.com/.

[7] D. Heimbigner. Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality. In *Proc. of the 2001 ACM Symposium on Applied Computing (SAC 2001)*, Las Vegas, Nevada, 11-14 March 2001.

[8] D. Heimbigner. Extending the Siena Publish/Subscribe Type System. Technical Report CU-CS-946-03, Department of Computer Science, University of Colorado, Jan. 2003.

[9] J. Hill. Site-Select Messaging for Distributed Systems. Technical Report CS-2002-06, Department of Computer Science, University of Virginia, Apr. 2002.

[10] *Jini^{tm} Specification, version 1.1 Beta*, 1999.

[11] A. M. Julienne and B. Holtz. *ToolTalk and open protocols, inter-application communication.* Prentice-Hall, 1994.

[12] *Kazaa Home Web Page*. http://kazaa.com/.

[13] J. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, and P. Devanbu. The Willow Survivability Architecture. In *Proc. of the Fourth Information Survivability Workshop*, Vancouver, B.C., March 18–20 2002.

[14] R. Lethin. Technical and social componentsof peer-to-peer computing. *CACM*, 46(2):30–32, 2002.

[15] *Mojo Nation Home Web Page*. http://www.mojonation.net/.

[16] *Napster Home Web Page*. http://www.napster.com/.

[17] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–67, July 1990.

[18] W. Segall and D. Arnold. Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching. In *Proceedings of the 1997 Australian UNIX Users Group*, Brisbane, Australia, Sept. 1997.

[19] TIBCO, Inc. *Rendezvous Information Bus*, 1996. http://www.rv.tibco.com/rvwhitepaper.html.

[20] M. van Renesse, K. Birman, and W. Vogels. Scalable Management, and Data Mining Using Astrolabe. In *Proc. of the 1st Int'l Workshop on Peer-to-Peer Systems*, Cambridge, Mass., Mar. 2002.

[21] J. Waldo. Jini^{tm} architectural overview: Technical white paper. Technical report, Sun Microsystems, 1999.

[22] J. Zien. The technology behind napster. *About*, 2000. http://internet.about.com/library/weekly/2000/aa052800b.htm.